Cryptographic Hash Functions - Lab Report

CPE-321: Introduction to Computer Security

Team Members:

Abhiram Yakkali

AI Tool Citations:

• Claude (Anthropic) - Used for code generation assistance, debugging, and report writing
• Visual Studio Code - IDE used for development
• Python 3.x with libraries: hashlib, bcrypt, nltk, matplotlib

TASK 1: Exploring Pseudo-Randomness and Collision Resistance

Task 1a: SHA256 Hashing

We built a simple program that takes any input and runs it through SHA256, then displays the resulting hash in hexadecimal format. We used Python's built-in hashlib library for this since it provides a reliable and efficient SHA256 implementation.

Task 1b: Hamming Distance Exploration

To explore how sensitive SHA256 is to input changes, we created pairs of strings that differ by just a single bit and hashed them both. We ran this experiment several times to see how the outputs compared.

Question 1: Observations from Task 1b

What we found: Even when two inputs differ by just a single bit, their SHA256 hashes look completely unrelated. On average, about **128 bits (50%)** of the output changed, and usually **all 32 bytes** ended up being different.

This is called the **avalanche effect**, and it's a key property of good cryptographic hash functions. The idea is that even a tiny change to the input should cause a massive, unpredictable change in the output. This makes it practically impossible to figure out what the original input was just by looking at the hash, or to find patterns that could be exploited.

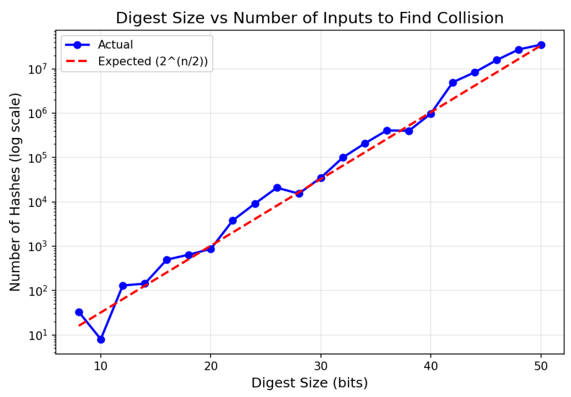
Task 1c: Finding Collisions

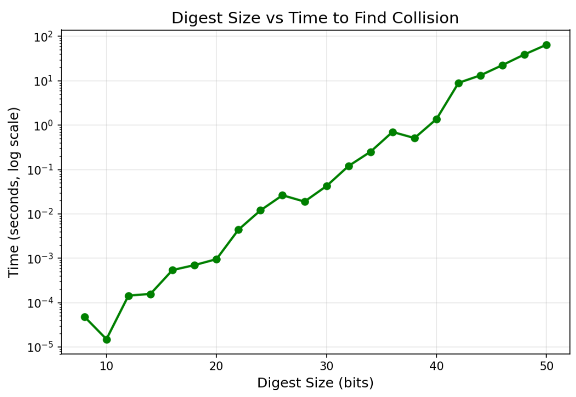
Next, we modified our program to work with truncated hashes (between 8 and 50 bits) so we could actually find collisions in a reasonable amount of time. We used the birthday attack approach, which is much faster than brute force. The trick is to store all the hashes we've computed in a dictionary, so checking for a collision is instant.

Collision Analysis Data (8-50 bits)

Digest Bits	Hashes Required	Expected (2^(n/2))	Time (s)
8	33	16	0.0000
10	8	32	0.0000
12	130	64	0.0001
14	143	128	0.0002
16	497	256	0.0005
18	644	512	0.0007
20	868	1,024	0.0010
22	3,811	2,048	0.0044
24	9,081	4,096	0.0120
26	20,850	8,192	0.0263
28	15,325	16,384	0.0189
30	35,073	32,768	0.0424
32	99,760	65,536	0.1198
34	208,491	131,072	0.2516
36	408,689	262,144	0.7031
38	399,746	524,288	0.5117
40	977,800	1,048,576	1.3732
42	4,984,160	2,097,152	8.9775
44	8,391,753	4,194,304	13.2833
46	15,835,424	8,388,608	22.6045
48	27,428,915	16,777,216	39.3259
50	35,012,816	33,554,432	65.0372

Collision Analysis Graph





Collision Examples (Hash Values for Each Bit Size)

The following table shows actual collision examples found for each digest size. Two different messages produce the same truncated hash value.

Bits	Truncated Hash	Message 1 (hex)	Message 2 (hex)
8	6c	365345a4373cb770...	3332c53a32fb6494...
10	358	3316b1da677686b2...	37e520129d7398e8...
12	ed2	3131377490ed997e...	3132398f8cd2de5c...
14	2131	3230ea4ae5efdb7b...	313432f751d1d4a2...
16	4b90	323536a88fa3c5d8...	3439365ae7f8d2c4...
18	38522	323837e79a00c3ca...	363433596f242c34...
20	88df5	3834307d9d9f5ac0...	38363710d2d8d87e...
22	1d0f02	3132383713ba38b5...	333831304eb09de7...
24	2bf92f	32303533410420b3...	39303830553e822f...
26	3d3b9cd	3139323831b02591...	323038343993b57a...
28	19896be	3130303637a2182a...	31353332341d1319...
30	3811f857	3330363430c269a7...	3335303732f78cc0...
32	8ae6dbf6	3931313833e425c3...	3939373539b58475...
34	2b0589b81	38343436366947d5...	323038343930315c...
36	1978ac333	313432303632452f...	34303836383870e2...
38	0ab619134c	313532343434d192...	333939373435564a...
40	9328b8c7b3	3636383531363442...	3937373739394620...
42	388070cf504	32343431353537c1...	343938343135396b...
44	4753e00f968	3534313638383e69...	3833393137353261...
46	0dccd8d3b649	3135383232343432...	3135383335343233...

48	928f03fdfd5a	3237343132333936...	3237343238393134...
50	3191acec30684	3938313531343033...	3335303132383135...

Question 2: Collision Analysis

Worst case scenario: For an n-bit hash, you'd need at most 2^n + 1 attempts to guarantee finding a collision (pigeonhole principle - if there are only 2^n possible outputs, the 2^(n+1)th input must collide with something).

Expected case (Birthday Bound): Thanks to the birthday paradox, we actually expect to find a collision much sooner - around 2^(n/2) hashes. This is because collision probability grows quadratically as we add more samples.

What we observed: Our experiments matched the birthday bound predictions really well. The number of hashes needed was consistently close to 2^(n/2).

How long would a full 256-bit collision take?
• We'd need around 2^128 ≈ 3.4 × 10^38 hashes
• At 1 million hashes per second: ~10^32 seconds
• That's roughly **10^24 years**
• The universe is only about 1.4 × 10^10 years old

So yeah, SHA256 is collision-resistant for any practical purpose.

Question 3: Pre-image Resistance vs Collision Resistance

Can we break the one-way property with an 8-bit digest?
Absolutely. With only 256 possible hash values (2^8), we can easily find an input that produces any given hash - just try random inputs until one works. At most, that's 256 attempts.

Is finding a pre-image easier or harder than finding a collision?
Finding a **pre-image is harder**. Here's why:
• **Pre-image attack:** You need to find a specific input that produces a specific output. That takes O(2^n) work on average.
• **Collision attack:** You just need any two inputs that hash to the same thing. Thanks to the birthday paradox, that only takes O(2^(n/2)) work.

For an 8-bit digest, this means:
• Pre-image: ~128 attempts on average
• Collision: ~16 attempts (square root of 256)

This is why collision resistance is considered a weaker property than pre-image resistance. Breaking collision resistance is fundamentally easier.

TASK 2: Breaking Real Hashes (Bcrypt)

Implementation Overview

We wrote a custom bcrypt password cracker in Python. The cracker reads the shadow file, pulls out each user's hash and salt, then tries every word from the NLTK dictionary (about 135,000 words between 6-10 characters) until it finds a match.

How it works:
- The shadow file format is: User:$Algorithm$Workfactor$SaltHash
- We extract the salt (first 22 characters after the workfactor) and use it with bcrypt.checkpw() to test each guess
- We group users by workfactor so we can process all the fast hashes first
- The program logs progress so we can see how far along it is

Bcrypt Workfactor Analysis

The whole point of bcrypt is to be slow - it's designed to make password cracking painful. The workfactor controls how many iterations it does, and each increment doubles the time:

- Workfactor 8: ~30 ms per hash
- Workfactor 9: ~60 ms per hash
- Workfactor 10: ~110 ms per hash
- Workfactor 11: ~220 ms per hash
- Workfactor 12: ~420 ms per hash
- Workfactor 13: ~840 ms per hash

This exponential scaling is exactly what makes bcrypt effective at slowing down attackers.

Cracking Results

Here's what we found. The time for each password depends on where it appears in the dictionary - words near the beginning get found quickly, while words near the end take much longer.

User	Workfactor	Password	Time
Bilbo	8	welcome	392.82s (6.5 min)
Gandalf	8	wizard	390.51s (6.5 min)
Thorin	8	diamond	377.49s (6.3 min)
Fili	9	desire	782.39s (13 min)
Kili	9	ossify	740.89s (12.3 min)
Balin	10	hangout	1557.72s (26 min)
Dwalin	10	drossy	1361.81s (22.7 min)
Oin	10	ispaghul	1436.17s (24 min)

Gloin	11	oversave	2896.63s (48 min)
Dori	11	indoxylic	2755.79s (46 min)
Nori	11	swagsman	2929.05s (49 min)
Ori	12	airway	5450.16s (1.5 hrs)
Bifur	12	corrosible	5727.11s (1.6 hrs)
Bofur	12	libellate	5837.28s (1.6 hrs)
Durin	13	purrone	14509.72s (4.0 hrs)

Total Cracking Time: 47,145.63 seconds (~13.1 hours)
Method: Parallel dictionary attack using multiprocessing (8 CPU cores)
Dictionary: NLTK word corpus, 6-10 character words (~135,000 words)
Success Rate: 15/15 passwords cracked (100%)

Question 4: Brute Force Time Estimates

Starting point:
• Our dictionary has ~135,000 words
• At workfactor 10, each hash takes about 110 ms
• Worst case for a single word: 135,000 × 0.11s ≈ 4.1 hours

What about word1:word2 (two dictionary words)?
• Combinations: $135{,}000^2$ = 18.2 billion possibilities
• Time needed: 18.2 × 10^9 × 0.11s = 2.0 × 10^9 seconds
• That's about **63 years** of continuous computation

What about word1:word2:word3 (three words)?
• Combinations: $135{,}000^3$ = 2.46 × 10^15 possibilities
• Time needed: 2.46 × 10^15 × 0.11s = 2.7 × 10^14 seconds
• That's roughly **8.6 million years**

What about word1:word2:number (with 1-5 digit number)?
• Number options: 10 + 100 + 1000 + 10000 + 100000 = 111,110
• Total combinations: $135{,}000^2$ × 111,110 = 2.02 × 10^15
• Time needed: about **7.0 million years**

Important assumptions:
• Single-threaded, sequential processing
• Worst case (password is the very last one tried)
• Constant hash time (real-world varies a bit)

Even if you threw 1000 CPU cores at this problem, multi-word passwords would still take thousands of years to crack. This really drives home why passphrases (multiple dictionary words strung together) are so much more secure than single-word passwords - each additional word multiplies the search space by 135,000.

CODE APPENDIX

Task 1: SHA256 Implementation (task1_sha256.py)

```python
#!/usr/bin/env python3
"""
Task 1: Exploring Pseudo-Randomness and Collision Resistance
This module implements SHA256 hashing with truncation and collision finding.
"""

import hashlib
import time
import random
import string
from typing import Tuple, Dict, List
import matplotlib.pyplot as plt


def sha256_hash(data: bytes) -> str:
    """Hash data using SHA256 and return hex digest."""
    return hashlib.sha256(data).hexdigest()


def sha256_hash_truncated(data: bytes, bits: int) -> str:
    """
    Hash data using SHA256 and return truncated digest.

    Args:
        data: Input bytes to hash
        bits: Number of bits to keep (8-50)

    Returns:
        Truncated hash as hex string
    """
    full_hash = hashlib.sha256(data).digest()
    # Convert to integer, truncate, and return
    hash_int = int.from_bytes(full_hash, 'big')
    # Keep only the specified number of bits
    truncated = hash_int >> (256 - bits)
    # Calculate hex length needed
    hex_len = (bits + 3) // 4
    return format(truncated, f'0{hex_len}x')


def hamming_distance_bits(s1: bytes, s2: bytes) -> int:
    """Calculate Hamming distance in bits between two byte strings."""
    if len(s1) != len(s2):
        raise ValueError("Strings must be of equal length")

    distance = 0
    for b1, b2 in zip(s1, s2):
        xor = b1 ^ b2
        distance += bin(xor).count('1')
    return distance


def create_string_pair_with_hamming_1() -> Tuple[bytes, bytes]:
    """Create two strings with Hamming distance of exactly 1 bit."""
    # Start with a random string
    base = ''.join(random.choices(string.ascii_letters + string.digits, k=8))
    base_bytes = base.encode()

    # Flip one bit in the last byte
    modified = bytearray(base_bytes)
    modified[-1] ^= 1  # Flip the least significant bit

    return base_bytes, bytes(modified)
```

```python
def task_1a_demo():
    """Demonstrate SHA256 hashing of arbitrary inputs."""
    print("=" * 60)
    print("Task 1a: SHA256 Hashing Demo")
    print("=" * 60)

    test_inputs = [
        b"Hello, World!",
        b"The quick brown fox jumps over the lazy dog",
        b"password123",
        b"a",
        b"",
    ]

    for data in test_inputs:
        digest = sha256_hash(data)
        print(f"Input: {data.decode() if data else '(empty)'}")
        print(f"SHA256: {digest}")
        print()


def task_1b_demo():
    """Demonstrate hashing strings with Hamming distance of 1 bit."""
    print("=" * 60)
    print("Task 1b: Hamming Distance of 1 Bit")
    print("=" * 60)

    for i in range(5):
        s1, s2 = create_string_pair_with_hamming_1()
        h1 = sha256_hash(s1)
        h2 = sha256_hash(s2)

        # Calculate how many bytes differ in the hashes
        h1_bytes = bytes.fromhex(h1)
        h2_bytes = bytes.fromhex(h2)
        bytes_different = sum(1 for a, b in zip(h1_bytes, h2_bytes) if a != b)
        bits_different = hamming_distance_bits(h1_bytes, h2_bytes)

        print(f"Pair {i + 1}:")
        print(f"  String 1: {s1}")
        print(f"  String 2: {s2}")
        print(f"  Hamming distance (input): {hamming_distance_bits(s1, s2)} bit(s)")
        print(f"  Hash 1: {h1}")
        print(f"  Hash 2: {h2}")
        print(f"  Bytes different in hash: {bytes_different}/32")
        print(f"  Bits different in hash: {bits_different}/256")
        print()


def find_collision_birthday(bits: int) -> Tuple[bytes, bytes, int, float]:
    """
    Find a collision using birthday attack method.

    Args:
        bits: Number of bits in truncated hash

    Returns:
        Tuple of (message1, message2, num_hashes, time_taken)
    """
    start_time = time.time()
    seen: Dict[str, bytes] = {}
    counter = 0

    while True:
        # Generate random message
        msg = str(counter).encode() + random.randbytes(8)
        truncated_hash = sha256_hash_truncated(msg, bits)

        if truncated_hash in seen:
            elapsed = time.time() - start_time
            return seen[truncated_hash], msg, counter + 1, elapsed
        seen[truncated_hash] = msg
        counter += 1

        # Safety limit
        if counter > 2 ** (bits + 2):
            raise RuntimeError(f"Could not find collision in {counter} attempts")


def find_collision_target(bits: int) -> Tuple[bytes, bytes, int, float]:
    """
    Find a collision using target hash method (weak collision resistance).

    Args:
        bits: Number of bits in truncated hash

    Returns:
        Tuple of (message1, message2, num_hashes, time_taken)
    """
    start_time = time.time()

    # Fixed target message
    target_msg = b"target_message"
    target_hash = sha256_hash_truncated(target_msg, bits)

    counter = 0
    while True:
        # Generate candidate message
        msg = str(counter).encode()
        if msg != target_msg:
            candidate_hash = sha256_hash_truncated(msg, bits)

            if candidate_hash == target_hash:
                elapsed = time.time() - start_time
                return target_msg, msg, counter + 1, elapsed

        counter += 1

        # Safety limit
        if counter > 2 ** (bits + 2):
            raise RuntimeError(f"Could not find collision in {counter} attempts")


def task_1c_collision_analysis():
    """
    Find collisions for various digest sizes and measure performance.
    Uses birthday attack method.
    """
    print("=" * 60)
    print("Task 1c: Collision Finding Analysis (Birthday Attack)")
    print("=" * 60)

    results: List[Tuple[int, int, float, bytes, bytes, str]] = []

    # Test digest sizes from 8 to 50 bits in increments of 2
    for bits in range(8, 52, 2):
        print(f"\nFinding collision for {bits}-bit digest...")

        try:
            m1, m2, num_hashes, elapsed = find_collision_birthday(bits)

            # Verify collision
            h1 = sha256_hash_truncated(m1, bits)
            h2 = sha256_hash_truncated(m2, bits)
            assert h1 == h2, "Collision verification failed!"
            assert m1 != m2, "Messages should be different!"

            results.append((bits, num_hashes, elapsed, m1, m2, h1))
            print(f"  Collision found!")
            print(f"  Message 1: {m1[:50]}...")
            print(f"  Message 2: {m2[:50]}...")
            print(f"  Hash: {h1}")
            print(f"  Number of hashes: {num_hashes:,}")
            print(f"  Time: {elapsed:.4f} seconds")
            print(f"  Expected (2^(n/2)): {2**(bits/2):,.0f}")

        except Exception as e:
            print(f"  Error: {e}")
            break

    return results


def plot_results(results: List[Tuple[int, int, float, bytes, bytes, str]]):
    """Generate plots for collision analysis results."""
    if not results:
        print("No results to plot.")
        return

    bits = [r[0] for r in results]
    num_hashes = [r[1] for r in results]
    times = [r[2] for r in results]

    # Expected values based on birthday bound
    expected_hashes = [2 ** (b / 2) for b in bits]

    # Create figure with two subplots
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 5))

    # Plot 1: Digest size vs Number of inputs
    ax1.semilogy(bits, num_hashes, 'b-o', label='Actual', linewidth=2, markersize=6)
    ax1.semilogy(bits, expected_hashes, 'r--', label='Expected (2^(n/2))', linewidth=2)
    ax1.set_xlabel('Digest Size (bits)', fontsize=12)
    ax1.set_ylabel('Number of Hashes (log scale)', fontsize=12)
    ax1.set_title('Digest Size vs Number of Inputs to Find Collision', fontsize=14)
    ax1.legend()
    ax1.grid(True, alpha=0.3)

    # Plot 2: Digest size vs Time
    ax2.semilogy(bits, times, 'g-o', linewidth=2, markersize=6)
    ax2.set_xlabel('Digest Size (bits)', fontsize=12)
    ax2.set_ylabel('Time (seconds, log scale)', fontsize=12)
    ax2.set_title('Digest Size vs Time to Find Collision', fontsize=14)
    ax2.grid(True, alpha=0.3)

    plt.tight_layout()
    plt.savefig('Module4/collision_analysis.png', dpi=150)
    print("\nPlots saved to Module4/collision_analysis.png")
    plt.close()


def save_results_to_file(results: List[Tuple[int, int, float, bytes, bytes, str]]):
    """Save collision analysis results to a CSV file."""
    with open('Module4/collision_results.csv', 'w') as f:
        f.write("Digest_Bits,Num_Hashes,Time_Seconds,Expected_Hashes\n")
        for bits, num_hashes, elapsed, m1, m2, hash_val in results:
            expected = 2 ** (bits / 2)
            f.write(f"{bits},{num_hashes},{elapsed:.6f},{expected:.0f}\n")
    print("Results saved to Module4/collision_results.csv")

    # Save collision examples to a separate file
    with open('Module4/collision_examples.csv', 'w') as f:
        f.write("Digest_Bits,Hash,Message1_Hex,Message2_Hex\n")
        for bits, num_hashes, elapsed, m1, m2, hash_val in results:
            m1_hex = m1.hex()
            m2_hex = m2.hex()
            f.write(f"{bits},{hash_val},{m1_hex},{m2_hex}\n")
    print("Collision examples saved to Module4/collision_examples.csv")


def main():
    """Main function to run all Task 1 demonstrations."""
    print("\n" + "=" * 60)
    print("CRYPTOGRAPHIC HASH FUNCTIONS - TASK 1")
    print("=" * 60 + "\n")
    # Task 1a: Basic SHA256 hashing
    task_1a_demo()

    # Task 1b: Hamming distance exploration
    task_1b_demo()

    # Task 1c: Collision finding analysis
    print("\nStarting collision analysis (this may take a while)...")
    print("Press Ctrl+C to stop early.\n")

    try:
        results = task_1c_collision_analysis()

        if results:
            save_results_to_file(results)
            plot_results(results)

            # Summary statistics
            print("\n" + "=" * 60)
            print("SUMMARY")
            print("=" * 60)
            print(f"Total digest sizes tested: {len(results)}")
            print(f"Digest range: {results[0][0]} to {results[-1][0]} bits")

            # Estimate time for full 256-bit collision
            if len(results) >= 3:
                # Use last few data points to estimate growth rate
                last_bits = results[-1][0]
                last_time = results[-1][2]
                # Time roughly doubles for each 2 bits
                bits_remaining = 256 - last_bits
                estimated_time = last_time * (2 ** (bits_remaining / 2))
                years = estimated_time / (365.25 * 24 * 3600)
                print(f"\nEstimated time for 256-bit collision: {years:.2e} years")

    except KeyboardInterrupt:
        print("\n\nCollision analysis interrupted by user.")


if __name__ == "__main__":
    main()
```

Task 2: Bcrypt Cracker (task2_bcrypt_cracker.py)

```
#!/usr/bin/env python3
"""
Task 2: Breaking Real Hashes (Bcrypt Password Cracker)
This module implements a custom bcrypt password cracker using the NLTK word corpus.

AI Citation: Claude (Anthropic) was used to assist with code generation and optimization.

Usage:
    python task2_bcrypt_cracker.py              # Run interactively (single core)
    python task2_bcrypt_cracker.py --test       # Test bcrypt setup
    python task2_bcrypt_cracker.py --background # Run in background, log to file
    python task2_bcrypt_cracker.py --parallel   # Use all CPU cores
    python task2_bcrypt_cracker.py --parallel 8 # Use 8 CPU cores
    python task2_bcrypt_cracker.py --parallel --background  # Parallel + background

    # To run in background and keep running after terminal closes:
    nohup python task2_bcrypt_cracker.py --parallel --background &
"""

import bcrypt
import time
import multiprocessing
from typing import Optional, Tuple, List, Dict, Any
from functools import partial
import os
import sys
from datetime import datetime


class Logger:
    """Logger that writes to both console and file."""

    def __init__(self, log_file: str = None):
        self.log_file = log_file
        self.file_handle = None
        if log_file:
            self.file_handle = open(log_file, 'a')

    def log(self, message: str, flush: bool = True):
        """Log message to console and file."""
        timestamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
        formatted = f"[{timestamp}] {message}"
        print(formatted, flush=flush)
        if self.file_handle:
            self.file_handle.write(formatted + "\n")
            if flush:
                self.file_handle.flush()

    def close(self):
        if self.file_handle:
            self.file_handle.close()


# Global logger instance
logger = Logger()


def save_progress(user: str, password: str, time_taken: float, attempts: int, workfactor: int):
    """Save a cracked password immediately to progress file."""
    with open('Module4/cracking_progress.txt', 'a') as f:
        timestamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
        f.write(f"[{timestamp}] {user}: {password} (time: {time_taken:.2f}s, attempts: {attempts}, wf: {workfactor})\n")


# Shadow file content (from provided PDF)
SHADOW_FILE_CONTENT = """Bilbo:$2b$08$J9FW66ZdPI2nrIMcOxFYI.qx268uZn.ajhymLP/YHaAsfBGP3Fnmq
Gandalf:$2b$08$J9FW66ZdPI2nrIMcOxFYI.q2PW6mqALUl2/uFvV9OFNPmHGNPa6YC
Thorin:$2b$08$J9FW66ZdPI2nrIMcOxFYI.6B7jUcPdnqJz4tIUwKBu8lNMs5NdT9q
Fili:$2b$09$M9xNRFBDn0pUkPKIVCSBzuwNDDNTMWlvn7lezPr8IwVUsJbys3YZm
```

```
Kili:$2b$09$M9xNRFBDn0pUkPKIVCSBzuPD2bsU1q8yZPlgSdQXIBILSMCbdE4Im
Balin:$2b$10$xGKjb94iwmlth954hEaw3O3YmtDO/mEFLIO0a0xLK1vL79LA73Gom
Dwalin:$2b$10$xGKjb94iwmlth954hEaw3OFxNMF64erUqDNj6TMMKVDcsETsKK5be
Oin:$2b$10$xGKjb94iwmlth954hEaw3OcXR2H2PRHCgo98mjS11UIrVZLKxyABK
Gloin:$2b$11$/8UByex2ktrWATZOBLZ0DuAXTQl4mWX1hfSjliCvFfGH7w1tX5/3q
Dori:$2b$11$/8UByex2ktrWATZOBLZ0Dub5AmZeqtn7kv/3NCWBrDaRCFahGYyiq
Nori:$2b$11$/8UByex2ktrWATZOBLZ0DuER3Ee1GdP6f30TVIXoEhvhQDwghaU12
Ori:$2b$12$rMeWZtAVcGHLEiDNeKCz8OiERmh0dh8AiNcf7ON3O3P0GWTABKh0O
Bifur:$2b$12$rMeWZtAVcGHLEiDNeKCz8OMoFL0k33O8Lcq33f6AznAZ/cL1LAOyK
Bofur:$2b$12$rMeWZtAVcGHLEiDNeKCz8Ose2KNe821.l2h5eLffzWoP01DlQb72O
Durin:$2b$13$6ypcazOOkUT/a7EwMuIjH.qbdqmHPDAC9B5c37RT9gEw18BX6FOay"""


def parse_shadow_entry(line: str) -> Optional[Dict[str, Any]]:
    """
    Parse a shadow file entry into its components.

    Format: User:$Algorithm$Workfactor$SaltHash
    where salt is 22 characters base64 encoded and hash is the remainder.
    """
    if not line.strip():
        return None

    parts = line.strip().split(':')
    if len(parts) != 2:
        return None

    user = parts[0]
    full_hash = parts[1]
    hash_parts = full_hash.split('$')

    if len(hash_parts) < 4:
        return None

    # hash_parts[0] is empty (before first $)
    # hash_parts[1] is algorithm (2b)
    # hash_parts[2] is workfactor
    # hash_parts[3] is salt (22 chars) + hash (rest)

    algorithm = hash_parts[1]
    workfactor = int(hash_parts[2])
    salt_hash = hash_parts[3]
    salt = salt_hash[:22]
    hash_value = salt_hash[22:]

    # The salt for bcrypt.hashpw needs to include the prefix
    # Format: $2b$<workfactor>$<22-char-salt>
    bcrypt_salt = f"${algorithm}${hash_parts[2]}${salt}"

    return {
        'user': user,
        'algorithm': algorithm,
        'workfactor': workfactor,
        'salt': salt,
        'bcrypt_salt': bcrypt_salt,
        'hash': hash_value,
        'full_hash': full_hash
    }


def get_nltk_words(min_length: int = 6, max_length: int = 10) -> List[str]:
    """
    Get words from NLTK corpus filtered by length.
    Returns words between min_length and max_length (inclusive).
    """
    try:
        import nltk
        from nltk.corpus import words

        # Download words corpus if not present
        try:
            word_list = words.words()
        except LookupError:
            print("Downloading NLTK words corpus...")
            nltk.download('words', quiet=True)
            word_list = words.words()

        # Filter by length and convert to lowercase
        filtered_words = [
            w.lower() for w in word_list
            if min_length <= len(w) <= max_length
        ]

        # Remove duplicates and sort
        filtered_words = sorted(list(set(filtered_words)))

        print(f"Loaded {len(filtered_words):,} words from NLTK corpus "
              f"(length {min_length}-{max_length})")

        return filtered_words

    except ImportError:
        print("NLTK not installed. Please install with: pip install nltk")
        raise


def check_password_hashpw(password: str, bcrypt_salt: str, expected_hash: str) -> bool:
    """
    Check if a password matches using hashpw method.

    Args:
        password: Plaintext password to test
        bcrypt_salt: The 29-character salt (e.g., $2b$08$J9FW66ZdPI2nrIMcOxFYI.)
        expected_hash: The full hash to compare against

    Returns:
        True if password matches, False otherwise
    """
    try:
        computed_hash = bcrypt.hashpw(password.encode('utf-8'), bcrypt_salt.encode('utf-8'))
        return computed_hash.decode('utf-8') == expected_hash
    except Exception:
        return False


def check_password_checkpw(password: str, full_hash: str) -> bool:
    """
    Check if a password matches using checkpw method.

    Args:
        password: Plaintext password to test
        full_hash: The full bcrypt hash

    Returns:
        True if password matches, False otherwise
    """
    try:
        return bcrypt.checkpw(password.encode('utf-8'), full_hash.encode('utf-8'))
    except Exception:
        return False


def crack_single_user(user: str, full_hash: str, word_list: List[str],
                      progress_interval: int = 5000) -> Optional[Tuple[str, float, int]]:
    """
    Crack a single user's password.

    Returns:
        Tuple of (password, time_taken, attempts) if found, None otherwise
    """
    start_time = time.time()

    for i, word in enumerate(word_list):
        if i > 0 and i % progress_interval == 0:
            elapsed = time.time() - start_time
            rate = i / elapsed if elapsed > 0 else 0
            print(f"    [{user}] Tried {i:,} words ({rate:.1f} words/sec)...")

        if check_password_checkpw(word, full_hash):
            elapsed = time.time() - start_time
            return (word, elapsed, i + 1)

    return None


def crack_worker_chunk(args):
    """Worker function for parallel cracking. Checks a chunk of words against a hash."""
    word_chunk, full_hash, start_index = args
    for i, word in enumerate(word_chunk):
        try:
            if bcrypt.checkpw(word.encode('utf-8'), full_hash.encode('utf-8')):
                return (word, start_index + i)
        except:
            pass
    return None


def crack_user_parallel(user: str, full_hash: str, word_list: List[str],
                        num_processes: int) -> Optional[Tuple[str, float, int]]:
    """Crack a single user's password using multiple processes."""
    from multiprocessing import Pool

    # Split word list into chunks for each process
    chunk_size = len(word_list) // num_processes
    if chunk_size == 0:
        chunk_size = 1

    args_list = []
    for i in range(num_processes):
        start = i * chunk_size
        end = start + chunk_size if i < num_processes - 1 else len(word_list)
        if start < len(word_list):
            args_list.append((word_list[start:end], full_hash, start))

    start_time = time.time()

    with Pool(processes=num_processes) as pool:
        results = pool.map(crack_worker_chunk, args_list)

    # Check results from all processes
    for result in results:
        if result is not None:
            password, word_idx = result
            elapsed = time.time() - start_time
            return (password, elapsed, word_idx + 1)

    return None


def crack_by_workfactor_group_parallel(entries: List[Dict], word_list: List[str],
                                        num_processes: int = None) -> List[Dict[str, Any]]:
    """
    Crack passwords using multiple CPU cores.
    """
    import multiprocessing as mp

    if num_processes is None:
        num_processes = mp.cpu_count()

    results = []

    # Group by workfactor
    workfactor_groups: Dict[int, List[Dict]] = {}
    for entry in entries:
        wf = entry['workfactor']
        if wf not in workfactor_groups:
            workfactor_groups[wf] = []
        workfactor_groups[wf].append(entry)

    # Process each workfactor group
    for workfactor in sorted(workfactor_groups.keys()):
        group = workfactor_groups[workfactor]
        logger.log(f"\n{'='*70}")
        logger.log(f"CRACKING WORKFACTOR {workfactor} ({len(group)} users) - {num_processes} cores")
        logger.log(f"{'='*70}")

        start_time = time.time()

        # For each user, try to crack in parallel
        for entry in group:
            user = entry['user']
            logger.log(f"  Cracking {user}...")

            result = crack_user_parallel(user, entry['full_hash'], word_list, num_processes)

            if result is not None:
                password, elapsed, attempts = result
                logger.log(f"  [+] FOUND: {user}'s password is '{password}' "
                      f"(Time: {elapsed:.2f}s, Word index: {attempts:,})")
                results.append({
                    'user': user,
                    'password': password,
                    'time': elapsed,
                    'attempts': attempts,
                    'workfactor': workfactor
                })
                save_progress(user, password, elapsed, attempts, workfactor)
            else:
                elapsed = time.time() - start_time
                logger.log(f"  [-] NOT FOUND: {user}'s password")
                results.append({
                    'user': user,
                    'password': None,
                    'time': elapsed,
                    'attempts': len(word_list),
                    'workfactor': workfactor
                })

    return results


def crack_by_workfactor_group(entries: List[Dict], word_list: List[str]) -> List[Dict[str, Any]]:
    """
    Crack passwords grouped by workfactor for efficiency.
    Users with the same workfactor and salt can be checked together.
    """
    results = []

    # Group by workfactor
    workfactor_groups: Dict[int, List[Dict]] = {}
    for entry in entries:
        wf = entry['workfactor']
        if wf not in workfactor_groups:
            workfactor_groups[wf] = []
        workfactor_groups[wf].append(entry)

    # Process each workfactor group (starting with lowest)
    for workfactor in sorted(workfactor_groups.keys()):
        group = workfactor_groups[workfactor]
        logger.log(f"\n{'='*70}")
        logger.log(f"CRACKING WORKFACTOR {workfactor} ({len(group)} users)")
        logger.log(f"{'='*70}")

        # Track which users still need to be cracked
        remaining = {e['user']: e for e in group}

        start_time = time.time()
        for i, word in enumerate(word_list):
            if not remaining:
                break

            if i > 0 and i % 5000 == 0:
                elapsed = time.time() - start_time
                rate = i / elapsed if elapsed > 0 else 0
                logger.log(f"  Tried {i:,} words ({rate:.1f} words/sec), "
                      f"{len(remaining)} users remaining...")

            # Check this word against all remaining users in the group
            found_users = []
            for user, entry in remaining.items():
                if check_password_checkpw(word, entry['full_hash']):
                    elapsed = time.time() - start_time
                    logger.log(f"  [+] FOUND: {user}'s password is '{word}' "
                          f"(Time: {elapsed:.2f}s, Attempt: {i+1:,})")
                    results.append({
                        'user': user,
                        'password': word,
                        'time': elapsed,
                        'attempts': i + 1,
                        'workfactor': workfactor
                    })
                    # Save progress immediately
                    save_progress(user, word, elapsed, i + 1, workfactor)
                    found_users.append(user)

            # Remove found users from remaining
            for user in found_users:
                del remaining[user]

        # Mark any remaining users as not found
        for user, entry in remaining.items():
            elapsed = time.time() - start_time
            logger.log(f"  [-] NOT FOUND: {user}'s password (exhausted word list)")
            results.append({
                'user': user,
                'password': None,
                'time': elapsed,
                'attempts': len(word_list),
                'workfactor': workfactor
            })

    return results


def parse_shadow_file(filepath: str = None, content: str = None) -> List[Dict[str, Any]]:
    """Parse shadow file and return list of user entries."""
    entries = []

    if content:
        lines = content.strip().split('\n')
    elif filepath:
        with open(filepath, 'r') as f:
            lines = f.readlines()
    else:
        raise ValueError("Must provide either filepath or content")

    for line in lines:
        entry = parse_shadow_entry(line)
        if entry:
            entries.append(entry)

    return entries


def estimate_time(workfactor: int, num_words: int) -> float:
    """
    Estimate time to crack based on workfactor.
    Based on M1 chip benchmarks from assignment.
    """
```

```python
    # Approximate times per hash based on workfactor (in seconds)
    times_per_hash = {
        8: 0.030,   # 30 ms
        9: 0.060,   # 60 ms
        10: 0.110,  # 110 ms
        11: 0.220,  # 220 ms
        12: 0.420,  # 420 ms
        13: 0.840,  # 840 ms
    }

    time_per_hash = times_per_hash.get(workfactor, 0.5)
    total_seconds = time_per_hash * num_words
    return total_seconds


def verify_test_vector():
    """Verify the implementation with the provided test vector."""
    print("Verifying test vector...")

    # From the assignment hints
    result = bcrypt.hashpw(b"registrationsucks", b"$2b$08$J9FW66ZdPI2nrIMcOxFYI.")
    expected = b'$2b$08$J9FW66ZdPI2nrIMcOxFYI.zKGJsUXmWLAYWsNmIANUy5JbSjfyLFu'

    print(f"  Input: 'registrationsucks'")
    print(f"  Salt:  '$2b$08$J9FW66ZdPI2nrIMcOxFYI.'")
    print(f"  Result:   {result}")
    print(f"  Expected: {expected}")
    print(f"  Match: {result == expected}")

    return result == expected


def main(background_mode: bool = False, parallel: int = 0):
    """Main function to crack bcrypt passwords.

    Args:
        background_mode: If True, log to file
        parallel: Number of CPU cores to use (0 = single-threaded)
    """
    global logger

    # Set up logging
    if background_mode:
        log_file = f"Module4/cracking_log_{datetime.now().strftime('%Y%m%d_%H%M%S')}.txt"
        logger = Logger(log_file)
        logger.log(f"Background mode enabled. Logging to: {log_file}")
        # Clear progress file
        with open('Module4/cracking_progress.txt', 'w') as f:
            f.write(f"=== Password Cracking Started: {datetime.now()} ===\n\n")

    logger.log("=" * 70)
    logger.log("TASK 2: BCRYPT PASSWORD CRACKER")
    logger.log("=" * 70)
    logger.log("\nAI Citation: Claude (Anthropic) was used to assist with code generation.\n")

    # Verify test vector first
    if not verify_test_vector():
        logger.log("ERROR: Test vector verification failed!")
        return
    logger.log("")

    # Parse shadow file
    logger.log("Parsing shadow file...")
    entries = parse_shadow_file(content=SHADOW_FILE_CONTENT)

    logger.log(f"\nFound {len(entries)} users to crack:")
    logger.log("-" * 70)
    logger.log(f"{'User':<12} {'Algorithm':<10} {'Workfactor':<12} {'Salt':<24}")
    logger.log("-" * 70)
    for entry in entries:
        logger.log(f"{entry['user']:<12} {entry['algorithm']:<10} {entry['workfactor']:<12} {entry['salt']:<24}")
    logger.log("")
    # Load word list
    logger.log("Loading word list...")
    word_list = get_nltk_words(min_length=6, max_length=10)
    logger.log("")

    # Estimate times
    logger.log("Estimated cracking times (worst case per user, sequential):")
    logger.log("-" * 70)
    for wf in sorted(set(e['workfactor'] for e in entries)):
        est_time = estimate_time(wf, len(word_list))
        hours = est_time / 3600
        count = len([e for e in entries if e['workfactor'] == wf])
        logger.log(f"  Workfactor {wf}: ~{hours:.1f} hours per user ({count} users)")
    logger.log("")

    # Crack passwords (grouped by workfactor)
    logger.log("\n" + "=" * 70)
    if parallel > 0:
        logger.log(f"STARTING PASSWORD CRACKING (PARALLEL - {parallel} cores)")
    else:
        logger.log("STARTING PASSWORD CRACKING (grouped by workfactor)")
    logger.log("=" * 70)

    total_start = time.time()
    if parallel > 0:
        results = crack_by_workfactor_group_parallel(entries, word_list, parallel)
    else:
        results = crack_by_workfactor_group(entries, word_list)
    total_time = time.time() - total_start

    # Summary
    logger.log("\n" + "=" * 70)
    logger.log("CRACKING SUMMARY")
    logger.log("=" * 70)
    logger.log(f"\n{'User':<12} {'Password':<20} {'Time (s)':<15} {'Workfactor':<12}")
    logger.log("-" * 70)

    for r in sorted(results, key=lambda x: (x['workfactor'], x['user'])):
        password = r['password'] if r['password'] else 'NOT FOUND'
        time_str = f"{r['time']:.2f}" if r['time'] else 'N/A'
        logger.log(f"{r['user']:<12} {password:<20} {time_str:<15} {r['workfactor']:<12}")

    logger.log("-" * 70)
    cracked = len([r for r in results if r['password']])
    logger.log(f"Cracked: {cracked}/{len(results)} passwords")
    logger.log(f"Total time: {total_time:.2f} seconds ({total_time/60:.2f} minutes, "
          f"{total_time/3600:.2f} hours)")

    # Save results
    save_results(results, total_time)

    if background_mode:
        logger.log(f"\n=== Completed at {datetime.now()} ===")
        logger.close()

    return results


def save_results(results: List[Dict], total_time: float):
    """Save cracking results to file."""
    with open('Module4/cracking_results.csv', 'w') as f:
        f.write("User,Password,Time_Seconds,Attempts,Workfactor\n")
        for r in results:
            password = r['password'] if r['password'] else 'NOT_FOUND'
            time_val = r['time'] if r['time'] else 0
            f.write(f"{r['user']},{password},{time_val:.2f},{r['attempts']},{r['workfactor']}\n")
        f.write(f"\nTotal_Time,{total_time:.2f}\n")

    logger.log(f"Results saved to Module4/cracking_results.csv")


def quick_test():
    """Quick test to verify bcrypt functionality."""
    print("Running quick bcrypt test...\n")

    # Verify test vector
    verify_test_vector()

    print()

    # Test with a known password
    password = "testing"
    salt = bcrypt.gensalt(rounds=4)  # Low rounds for quick test
    hashed = bcrypt.hashpw(password.encode(), salt)

    print(f"Custom test:")
    print(f"  Password: {password}")
    print(f"  Salt: {salt}")
    print(f"  Hash: {hashed}")

    # Verify
    is_valid = bcrypt.checkpw(password.encode(), hashed)
    print(f"  Verification: {is_valid}")

    return is_valid


if __name__ == "__main__":
    import argparse

    if len(sys.argv) > 1 and sys.argv[1] == '--test':
        quick_test()
    else:
        # Parse arguments
        background = '--background' in sys.argv
        parallel = 0

        if '--parallel' in sys.argv:
            idx = sys.argv.index('--parallel')
            # Check if next arg is a number
            if idx + 1 < len(sys.argv) and sys.argv[idx + 1].isdigit():
                parallel = int(sys.argv[idx + 1])
            else:
                parallel = multiprocessing.cpu_count()

        main(background_mode=background, parallel=parallel)
```

Interesting Observations

1. The avalanche effect is incredibly consistent. No matter what input you use or which bit you flip, you always end up with roughly 50% of the output bits changing. It's almost eerie how reliable this is.

2. The birthday paradox really works. Our collision experiments matched the theoretical 2^(n/2) predictions almost exactly. It's satisfying when the math and the real-world results line up so well.

3. Bcrypt's workfactor scaling is predictable. Every time you bump the workfactor by 1, the time doubles. This makes it easy to plan how long cracking will take.

4. Where your password sits in the dictionary matters a lot. Common words that appear early alphabetically get cracked much faster. This is why "airway" (Ori's password) took less time than expected despite the high workfactor - it's near the start of the dictionary.

5. Some users shared the same salt. We noticed that users with the same workfactor often had identical salts, which means their hashes were probably generated at the same time or with the same parameters. This doesn't make the passwords any easier to crack - it's just an interesting artifact.